Lecture Notes on
Memory Optimizations

15-411: Compiler Design
Frank Pfenning and Jan Hoffmann

Lecture 18
November 1, 2016

1 Introduction

Even on modern architecures with hierarchical memory caches, memory access, on average, is still significantly more expensive than register access or even most arithmetic operations. Therefore, memory optimizations play a significant role in generating fast code. As we will see, whether certain memory optimizations are possible or not depends on properties of the whole language. For example, whether or not we can obtain pointers to the middle of heap-allocated objects will be a crucial question to answer.

2 A Simple Example

We will use a simple running example to illustrate memory optimization and their conditions of applicability. In this example, $\mathsf{mult}(A, p, q)$ will multiply matrix A with vector p and return the result in vector q.

```
struct point {
  int x;
  int y;
};
typedef struct point pt;

void mult(int[] A, pt* p, pt* q) {
  q->x = A[0] * p->x + A[1] * p->y;
  q->y = A[2] * p->x + A[3] * p->y;
  return;
}
```

Below is the translation into abstract assembly, with the small twist that we have allowed memory reference to be of the form $M[\mathit{base} + \mathit{offset}]$. The memory optimization question we investigate is whether some load instructions $t \leftarrow M[s]$ can be avoided because the corresponding value is already held in a temp.

```
mult$(A, p, q)$ :
    $t_0 \leftarrow M[A + 0]$
    $t_1 \leftarrow M[p + 0]$
    $t_2 \leftarrow t_0 * t_1$
    $t_3 \leftarrow M[A + 4]$
    $t_4 \leftarrow M[p + 4]$
    $t_5 \leftarrow t_3 * t_4$
    $t_6 \leftarrow t_2 + t_5$
    $M[q + 0] \leftarrow t_6$
    $t_8 \leftarrow M[A + 8]$
    $t_9 \leftarrow M[p + 0]$        # redundant load?
    $t_{10} \leftarrow t_8 * t_9$
    $t_{11} \leftarrow M[A + 12]$
    $t_{12} \leftarrow M[p + 4]$     # redundant load?
    $t_{13} \leftarrow t_{11} * t_{12}$
    $t_{14} \leftarrow t_{10} + t_{13}$
    $M[q + 4] \leftarrow t_{14}$
    return
```

We see that the source refers to `p->x` and `p->y` twice, and those are reflected in the two, potentially redundant loads above. Before you read on, consider if we could replace the lines with $t_9 \leftarrow t_1$ and $t_{12} \leftarrow t_4$. We can do that if we can be assured that memory at the addresses $p + 0$ and $p + 4$, respectively, has not changed since the previous load instructions.

It turns out that in C0 the second load is definitely redundant, but the first one may not be.

The first load is not redundant because when this function is called, the pointers p and q might be the same (they might *aliased*). When this is the case, the store to $M[q{+}0]$ will likely change the value stored at $M[p{+}0]$, leading to a different answer than expected for the second line.

On the other hand, this cannot happen for the first line, because $M[q+0]$ could never be the same as $M[p+4]$ since one accesses the x field and the other the y field of a struct.

Of course, the answer is mostly likely wrong when $p = q$. One could either rewrite the code, or require that $p \neq q$ in the precondition to the function.

In C, the question is more delicate because the use of the address-of (`&`) operator could obtain pointers to the middle of objects. For example, the argument `int[] A` would be `int* A` in C, and such a pointer might have been obtained with `&q->x`.

3 Using the Results of Alias Analysis

In C0, the types of pointers are a powerful basis of alias analysis. The way alias analysis is usually phrased is as a *may-alias* analysis, because we try to infer which pointers in a program may alias. Then we know for optimization purposes that if two pointers are not in the *may-alias* relationship that they must be different. Writing to one address cannot change the value stored at the other.

Let's consider how we might use the results of alias analysis, embodied in a predicate $\mathsf{may\text{-}alias}(a, b)$ for two addresses a and b. We assume we have a load instruction

$$l : t \leftarrow M[a]$$

and we want to infer if this is *available* at some other line $l' : t' \leftarrow M[a]$ so we could replace it with $l' : t' \leftarrow t$. Our optimization rule (in the notation of linear inference from Lecture 16):

$$\left.\begin{array}{l} l : t \leftarrow M[a] \\ \ldots \\ k : t' \leftarrow M[a] \end{array}\right\} \longrightarrow \left\{\begin{array}{l} l : t \leftarrow M[a] \\ \ldots \\ k : t' \leftarrow t \end{array}\right. \qquad \text{provided} \quad l > k, \mathsf{avail}(l, k)$$

The fact that l dominates k is sufficient here in SSA form to guarantee that the meaning of t and a remains unchanged. avail is supposed to check that $M[a]$ also remains unchanged.

Reaching analysis for memory references is a simple forward dataflow analysis. If we have a node with two or more incoming control flow edges, it must be available along all of them. For the purposes of traversing loops we assume availability, essentially trying to find a counterexample in the loop. To express this concisely, our analysis rules propagate *unavailability* of a definition $l : t \leftarrow M[a]$ at other instructions k that are dominated by l.

For unavailability, $\mathsf{unavail}(l, k)$, we have the seeding rule on the left and the general propagation rule on the right. Because we are in SSA, we know in the seeding rule that $l > k$ where k is the (unique) successor of l'.

$$\frac{\begin{array}{l} l : t \leftarrow M[a] \\ l > l' \\ l' : M[b] \leftarrow s \\ \mathsf{may\text{-}alias}(a, b) \\ \mathsf{succ}(l', k) \end{array}}{\mathsf{unavail}(l, k)} \qquad \frac{\begin{array}{l} \mathsf{unavail}(l, k) \\ \mathsf{succ}(k, k') \\ l > k' \end{array}}{\mathsf{unavail}(l, k')}$$

The rule on the right includes the cases of jumps or conditional jumps. This ensures that in a node with multiple predecessors, if a value is unavailable in just one of them, in will be unavailable at the node. Function calls can also seed unavailability. Unfortunately it is enough if one of the function parameters is a memory reference, because from one memory reference we may be able to get to another by following pointers and offsets.

$$\frac{\begin{array}{l} l : t \leftarrow M[a] \\ l > l' \\ l' : d \leftarrow f(s_1, \ldots, s_n) \\ \mathsf{memref}(s_i) \\ \mathsf{succ}(l', k) \end{array}}{\mathsf{unavail}(l, k)}$$

With more information on the shape of memory this rule can be relaxed.

From unavailability we can deduce which memory values are still available, namely those that are not unavailable (restriction attention to those that are dominated by the load—otherwise the question is not asked).

$$\frac{\begin{array}{l} l : t \leftarrow M[a] \\ l > l' \\ \neg\mathsf{unavail}(l, l') \end{array}}{\mathsf{avail}(l, l')}$$

Note that stratification is required: we need to saturate $\mathsf{unavail}(l, l')$ before applying this rule.

4 Type-Based Alias Analysis

One way of performing type-based alias analysis is to propagate type information from the semantic analysis to abstract assembly. The type information is used to associate with each memory location the type of data that is stored at that location. The idea is that data that is stored at locations of different types is not aliased. Such an analysis works for type-safe languages such as Java and C0.

Here, we will restore the type information directly at the assembly level. Our alias analysis is based on the type and offset of the address. We call this an *alias class*, with the idea that pointers in different alias classes cannot alias. More formally, we derive a predicate $\mathsf{class}(a, \tau, \mathit{offset})$, which expresses that a is an address derived from a source of type τ and offset offset from the start of the memory of type τ.

Then the *may-alias* relation is defined by

$$\frac{\mathsf{class}(a, \tau, k) \quad \mathsf{class}(b, \tau, k)}{\mathsf{may\text{-}alias}(a, b)}$$

There is a couple of special cases we do not treat explicitly. For example, the location of the array length (which is stored in safe mode at least) may be at offset -8. But such a location can never be written to (array lengths never change, once allocated), so a load of the array length is available at all locations dominated by the load.

The seed of the class relation comes from function types and need to be propagated by the compiler. In our example,

```
$\mathsf{mult}(A, p, q)$ :
    $t_0 \leftarrow M[A + 0]$
    $t_1 \leftarrow M[p + 0]$
    $t_2 \leftarrow t_0 + t_1$
    $t_3 \leftarrow M[A + 4]$
    $\ldots$
```

the compiler would generate

$$\begin{array}{l}\mathsf{class}(A, \mathsf{int}[\,], 0)\\ \mathsf{class}(p, \mathsf{struct\ point}{*}, 0)\\ \mathsf{class}(q, \mathsf{struct\ point}{*}, 0)\end{array}$$

We now propagate the information through a forward dataflow analysis. For example:

$$\frac{l : b \leftarrow a \quad \mathsf{class}(a, \tau, k)}{\mathsf{class}(b, \tau, k)} \qquad \frac{l : b \leftarrow a + \$n \quad \mathsf{class}(a, \tau, k)}{\mathsf{class}(b, \tau, k + n)}$$

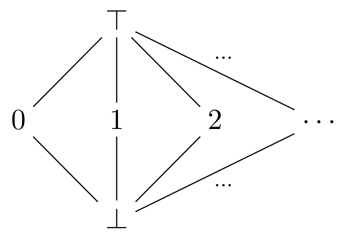
In the second case we have written $\$n$ to emphasize the second summand is a constant n. Unfortunately, if it is a variable, we cannot precisely calculate the offset. This may happen with arrays, but not with pointers, including pointers to structs.

So we need to generalize the third argument to class to be either a variable or $\top$, which indicates any value may be possible. We then have, for example

$$\frac{l : b \leftarrow a + t \quad \mathsf{class}(a, \tau, k)}{\mathsf{class}(b, \tau, \top)}$$

Now $\top$ behaves like an information sink. For example, $\top + k = k + \top = \top$. Since in SSA form a is defined only once, we should not have to change our mind about the class assigned to a variable. However, at parameterized jump targets (which is equivalent to Φ-functions), we need to "disjoin" the information so that if the argument is known to be k at one predecessor but unknown at $\top$ at another predecessor, the result should be $\top$.

Because of loops, we then need to generalize further and introduce $\bot$ which means that we believe (for now) that the variable is never used. Because of the seeding by the compiler, this will mostly happen for loop variables. The values are arranged in a *lattice*



where at the bottom we have more information, at the top the least. The $\sqcup$ operation between lattice elements finds the least upper bounds of its two arguments. For example, $0 \sqcup 4 = \top$ and $\bot \sqcup 2 = 2$. We use it in SSA form to combine information about offsets. We now read an assertion $\mathsf{class}(a, \tau, k)$ as saying that the offset is at least k under the lattice ordering. Then we have

$$\frac{\begin{array}{l} \mathsf{lab}(a_1) : \\ \mathsf{class}(a_1, \tau, k_1) \\ l : \mathsf{goto}\ \mathsf{lab}(a_2) \\ \mathsf{class}(a_2, \tau, k_2) \end{array}}{\mathsf{class}(a_1, \tau, k_1 \sqcup k_2)}$$

Written with Φ-functions, we would have

$$\frac{\begin{array}{l} \mathsf{class}(a_0, \tau, k_0) \\ a_0 \leftarrow \Phi(a_1, \ldots, a_n) \\ \mathsf{class}(a_i, \tau, k_i) \quad (1 \leq k \leq n) \end{array}}{\mathsf{class}(a_0, \tau, k_0 \sqcup k_1 \sqcup \cdots \sqcup k_n)}$$

Because of loops we might perform this calculation multiple times until we have reached a fixed point. In this case the fixed point is least upper bound of all the offset classes we compute, which is a little different than the saturated data base we considered before.

It is not immediately clear why the different classes of the addresses a_i in the last two rules have the same type τ. For example, it could be possible that we have $\mathsf{class}(a_1, \tau_1, k_1)$ and $\mathsf{class}(a_2, \tau_2, k_2)$ for $\tau_1 \neq \tau_2$ in the first rule. Whether this is indeed possible or not depends on the translations and optimizations that are performed by your compiler. If such cases happen then you also need to introduce a lattice for the types and for instance use the join $\tau_1 \sqcup \tau_2 = \top$.

This is an example of *abstract interpretation*, which may be a subject of a future lecture. One can obtain a more precise alias analysis if one refines the *abstract domain*, which is lattice shown above.

5 Allocation-Based Alias Analysis

Another technique to infer that pointers may not alias is based on their allocation point. In brief, if two pointers are allocated with different calls to alloc or alloc_array, then they cannot be aliased. Because allocation may happen in a different function than we are currently compiling (and hopefully optimizing), this is an example of an *interprocedural analysis*.

References